Tampereen ammattikorkeakoulu

2D-PELIMOOTTORI

Miika Mattila

OPINNÄYTETYÖ
Huhtikuu 2023

Tietoliikenne- ja viestintätekniikka
Tietoliikennetekniikka ja tietoverkot

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietoliikenne- ja viestintätekniikka
Tietoliikennetekniikka ja tietoverkot

MATTILA, MIIKA:
2D-PELIMOOTTORI

Opinnäytetyö 39 sivua, joista liitteitä 1 sivu
Huhtikuu 2023

Pelimoottori on käsite, joka syntyi 1990-luvulla. Pelimoottori pitää sisällään pelikehityksessä käytettyjä yhteisiä komponentteja ja työkaluja. Yksi ensimmäisistä pelimoottoreista oli id Softwaren kehittämä ”DOOM engine” Doom -pelisarjaa varten.

Pelimoottorin isoimmat kokonaisuudet koostuvat pelisilmukasta, sekä kuvanpiirto-, animaatio-, fysiikka- ja äänijärjestelmästä. Pelisilmukka on koko pelimoottorin ja pelin ydin, joka pitää ohjelman käynnissä sekä päivittää muita järjestelmiä tietyn aikavälin jälkeen, jolloin pelin tapahtumat tapahtuvat yhdenmukaisesti.

Tämä opinnäytetyö keskittyy 2D-pelimoottorin kehittämiseen, joka tarjoaa yksinkertaisen pohjan pelien kehittämiseen. Pelimoottori sisältää animaatio-, törmäyksen tunnistus-, maailmojen lataus- ja yksinkertaisen ECS-järjestelmän. Lisäksi pelimoottorin toimintoja testattiin luomalla testikohtauksia, joissa päästiin testaamaan kuvanpiirto-, törmäys- ja maailmojen latausjärjestelmien toimivuutta.

Asiasanat: tietokonepeli, pelimoottori, 2d

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Telecommunications and communication technology
Telecommunication technology and data networks

MATTILA, MIIKA:
2D GAME ENGINE

Bachelor's thesis 39 pages, appendices 1 pages
April 2023

The term game engine was born in the 1990s and one of the first game engines was DOOM engine that was developed by id Software for Doom game series.

Game engine consists of common components and tools that are used for game development. Such as game loop, rendering system, animation system, physics system and sound system. In the core of all real time applications such as a game is game loop. Game loop keeps the game running and updating the systems of a game with a certain time interval to keep game events occurring consistently.

The focus of this thesis is the development of a simple 2D game engine that will provide the basic systems to develop a game. These basic systems include rendering system, animation system, map parsing system and collision system.

Key words: computer game, game engine, 2d

SISÄLLYS

ERITYISSANASTO

2D	Kaksiulotteinen (Two Dimension)
bittimaski	Määrittelee, mitkä bittijonon bittien arvot ovat olennaisia ja mitkä eivät
dt	Delta-aika, s (sekunnit); muuttuja, joka kertoo kahden kuvakehyksen välinen ajan
ECS	Entity Component System; ohjelmiston arkkitehtuurimalli
Entity	Abstrakti objekti; tässä tapauksessa pelin elementti, ikoni, symboli tai peliolio
FPS	First Person Shooter; peligenre tai pelityyppi
fps	Kuvakehysten määrä sekunnissa (frames per second)
frame	Yksittäinen kuva videossa; liikkuvan kuvan ruutu
Objekti	Ohjelman itsenäisiä osia, jotka on luotu luokkien avulla ja sisältävät ominaisuuksia (muuttujia) ja toiminnallisuuksia (metodeja)
SFML	Simple and Fast Multimedia Library; C++-pohjainen multimediakirjasto, joka tarjoaa OpenGL-pohjaisen ikkunan sisällön käsittelyn
Texture	Tekstuuri, pintakuviointi
Sprite	2D-kuva, jolla on sijainti pelimaailmassa
Spritesheet	Kuva, joka sisältää kokoelman eri kuvista

1 JOHDANTO

2000-luvulla peliteollisuus on noussut merkittäväksi alaksi, ja tietokonepelit ovat kehittyneet huomattavasti alkuaikojen kotikonsolien (kuten Odyssey, Atari 2600 ja NES) ja kotitietokoneiden (kuten Radio Shack TRS-80 Sinclair ZX80, Spectravideo, VIC20 ja Commodore 64) jälkeen.

Pelien grafiikan, suorituskyvyn ja käyttäjäystävällisten ohjaustapojen realistisuuden tavoittelu on saavuttanut yhä korkeampia tasoja, esimerkiksi Microsoft Xbox Kinectin 3D-tilanhavainnoinnin tai 3D VR-lasien avulla. Internetin ansiosta yhteispelaaminen on laajentunut maailmanlaajuisesti, mahdollistaen yksilöiden ja yhteisöjen pelaamisen. Kehityksen suuntaus näyttää olevan kohti yhä realistisempaa pelikokemusta ja täydellisemmän virtuaalitodellisuuden saavuttamista.

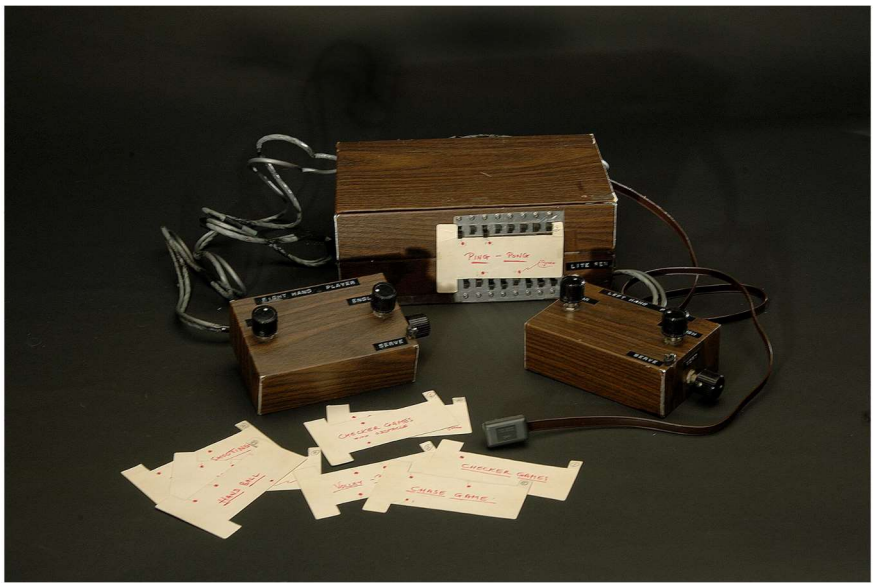
Ohjelmakehitysympäristöjen kehittyminen alkuaikojen laitteistopohjaisesta ohjelmoinnista korkeammalle tasolle on mahdollistanut nopeamman ohjelmistokehityksen ja virheiden löytämisen. Pelimoottorit ja niiden hyödyntäminen ovat keino nopeuttaa pelien valmistumista luomalla alustoja ja ohjelmistokirjastoja. Kaupallisten tuotteiden valmistaminen vaatii tehokkuutta ja nopeaa markkinoille tuloa.

Tämän työn toteutuksessa on seurattu Robert Wellsin tuottamaa 2D-pelimoottorin kehitys sarjaa CPP Game Dev. Tämän pohjalta saatiin aikaiseksi toimiva 2D-pelimoottori, johon sisältyy animaatiojärjestelmä, törmäyksen tunnistusjärjestelmä, maailmojen latausjärjestelmä sekä yksinkertainen ECS-järjestelmä. Nämä osa-alueet yhdessä muodostavat kokonaisvaltaisen ratkaisun pelien kehittämiseen ja tarjoavat vankan pohjan jatkokehitykselle.

2 PELIKEHITYKSEN HISTORIAA

Pelikehityksen historia kattaa yli viiden vuosikymmenen ajanjakson aina 1950-luvulle asti, jolloin ensimmäiset videopelit otettiin käyttöön. Yksi varhaisimmista videopeleistä oli William Higinbothamin luoma Tennis for Two (The First Video Game? 2008).

The Brown Box (kuva 1) oli yksi ensimmäisistä videopelikonsoleista, jonka Ralph Baer kehitti työskennellessään Sanders Associates, Inc:lle. Tällä konsolilla oli mahdollista pelata useita eri pelejä sekä moninpelipelejä. (Video Game History 2022).



Kuva 1. The Brown Box (The Brown Box, 1967-68. 2006).

Alussa pelit koodattiin joko suoraan laitteistoon tai ne luotiin assembly-ohjelmoinnin avulla, jotta saavutettaisiin mahdollisimman tehokas hyödyntäminen rajoitetusta prosessointitehosta ja muistista.

Tämä lähestymistapa sopi yksinkertaisille peleille, lisääntyneen prosessointitehon ja muistin myötä pelinkehittäjät alkoivat suunnitella monimutkaisempia pelejä. Tässä vaiheessa korkeamman tason ohjelmointikielet, kuten C/C++, tulivat avuksi.

Pelikehityksessä huomattiin, että monet pelit tarvitsivat samankaltaisia ohjelmointirakenteita ja tekniikoita. Tästä seurasi pelimoottorien tarve, joka yhdistää nämä usein käytetyt elementit yhdeksi pelikehitysympäristöksi.

Yksi ensimmäisistä pelimoottoreista oli id Softwaren kehittämä "DOOM engine" DOOM-pelisarjaa (kuva 2) varten (Game Engines and History 2023).

Kuva 2. DOOM-pelin näyttökuva (Lowood 2020).

3 PELIMOOTTORIT

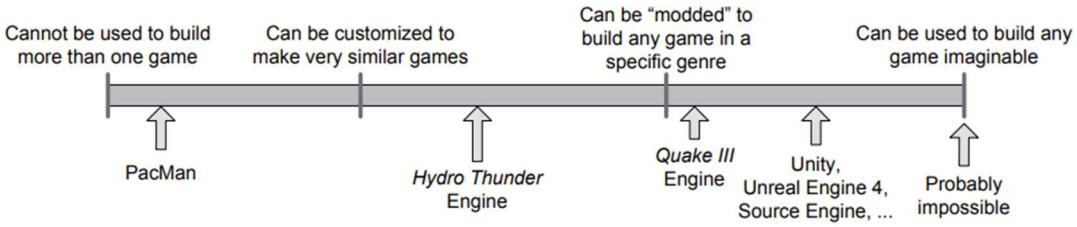
Tässä luvussa käydään läpi muutamia yleisimpiä pelimoottoreita kuten Unity, Cryengine ja Source engine. Tämän lisäksi esitellään pelimoottorin järjestelmiä kuten pelisilmukka, kuvanpiirto-, fysiikka-, animaatio-, ääni- ja ECS-järjestelmä.

3.1 Yleisimpiä pelimoottoreita

Pelimoottori on ohjelmisto, jolla on tarkoitus helpottaa pelien kehittämistä ja tuottamista. Se tarjoaa valmiita työkaluja ja rajapintoja, jotka auttavat pelinkehittäjiä toteuttamaan pelinsä ilman, että he joutuvat luomaan kaikkea alusta alkaen itse.

Pelimoottorin ideana on erotella ohjelmiston ydinosaset itse pelin osa-alueista. Esimerkiksi yhdessä pelissä kuvanpiirtojärjestelmä ”tietää” kuinka piirtää peikon. Kuin taas toisen pelin kuvanpiirtojärjestelmä voisi tarjota yleiskäyttöisen materiaali ja varjostinpalvelun ja peikkomaisuus määritellään datan avulla. (Gregory 2018, 12).

Dataohjattuna arkkitehtuurina pidetään sellaista peliarkkitehtuuria, jossa pelimoottori on selkeästi eroteltu itse pelistä. Ilman modulaarista rakennetta peli sisältää kovakoodattua logiikkaa tai pelisääntöjä. Tämänlaisen ohjelmiston uudelleenkäyttö toisen pelin tekemiseen on vaikeaa tai mahdotonta. (Gregory 2018, 12).
Modulaarisen rakenteisen (esim. Unity) ja kovakoodatun pelin (esim. PacMan) määrittely ei ole mustavalkoista, vaan jokaista peliä voidaan tarkastella uudelleenkäyttöasteikolla modulaarisuuden suhteen. (Kuva 3).



Kuva 3. Pelimoottorin uudelleenkäyttöasteikko (Gregory 2018, 12).

Pelimoottoreita on useita erilaisia ja ne tarjoavat erilaisia työkaluja sekä rajapintoja erilaisiin tarpeisiin. Esimerkiksi jotkut pelimoottorit ovat erikoistuneita 2D-pelien tekemiseen, kun taas toiset ovat parempia 3D-pelien tekemiseen. Jotkut pelimoottorit ovat myös erikoistuneet tiettyihin genreihin, kuten esimerkiksi strategiapeleihin tai urheilupeleihin.

Pelimoottorit tarjoavat usein valmiita toiminnallisuuksia, kuten kameran liikuttelun, hahmojen liikkumisen ja törmäyksien tunnistamisen. Ne sisältävät myös usein grafiikka- ja fysiikkamoottoreita, äänen ja musiikin toisto- ja synkronointityökaluja sekä tekstuuri- ja mallinnustyökaluja. Pelimoottorilla on myös usein tarvittavat rajapinnat pelin kontrollien ja pelaajan syötteen käsittelyyn.

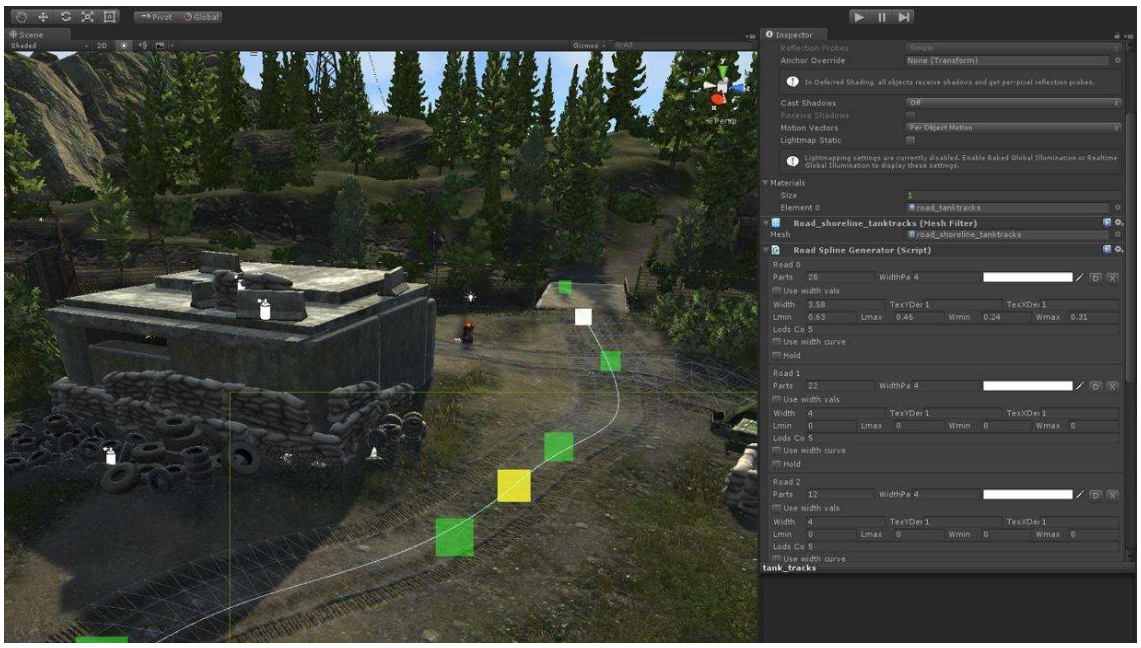
Seuraavissa osissa esitellään kaksi suosittua pelimoottoria (Tyler 2023) ja kolmantena pelimoottori, jonka avulla on luotu pelejä, joilla on miljoonia päivittäisiä pelaajia. (Steam Charts 2023).

3.1.1 Unity

Unity on pelimoottori, joka on suunniteltu auttamaan kehittäjiä luomaan pelejä ja interaktiivisia sovelluksia. Se tarjoaa useita työkaluja ja tekniikoita, jotka helpottavat ohjelmointia ja visualisointia. Se on saatavana useille eri alustoille, kuten PC:lle, Macille, mobiililaitteille ja verkkoselaimille.

Se tarjoaa myös kattavan dokumentaation ja yhteisön, josta kehittäjät voivat saada apua ja tukea pelien ja sovellusten kehittämiseen liittyvissä kysymyksissä.

Esimerkkinä Unityllä kehitetystä pelistä on Escape from Tarkov, joka on Battlestate Gamesin kehittämä ensimmäisen persoonan ampumispeli (kuva 4).



Kuva 4. Escape from Tarkov -pelin kehityskuva (Unity Escape from Tarkov 2023).

3.1.2 Cryengine

Cryengine on pelimoottori, joka on suunniteltu erityisesti korkean tarkkuuden ja näyttävien grafiikan toteuttamiseen. Se on kehitetty Crytek-yhtiön toimesta ja se on alun perin suunniteltu erityisesti Far Cry -pelisarjalle. Cryengine on kuitenkin laajentanut käyttökenttäänsä ja sitä voidaan käyttää myös muiden pelien kehittämiseen. (Cryengine. 2023).

Cryengine tarjoaa monipuoliset työkalut pelin suunnitteluun, kuten 3D-mallinnukseen, animaatioon ja teksturointiin. Se sisältää myös monia valmiita komponentteja, kuten fysiikkamoottorin ja AI-järjestelmän, jotka helpottavat pelin kehittämistä. Cryengine on saatavilla useille eri alustoille, kuten PC:lle, konsolille ja mobiililaitteille. (Cryengine 2023).

Esimerkkinä Cryenginellä tehdystä pelistä on Crysis pelisarja (kuva 4). Tämä pelisarja on edistänyt graafista todentuntuisuutta.

Kuva 5. Crysis 3 (Cryengine Crysis 3 2023).

3.1.3 Source Engine

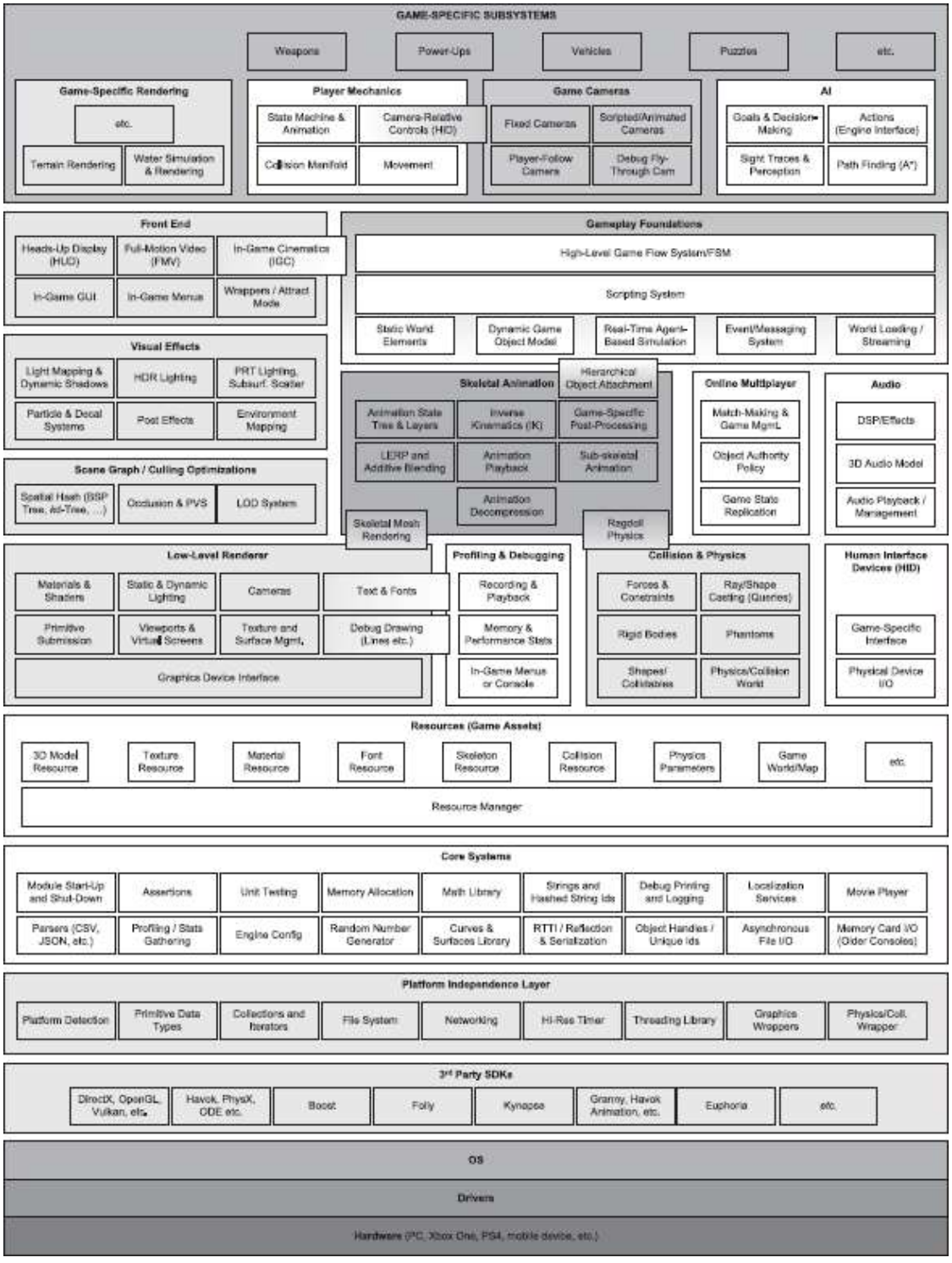
Source Engine on pelimoottori, jonka kehitti Valve Corporatioinin vuonna 2004. Moottori on suunniteltu erityisesti tietokoneille ja tarjoaa monia erilaisia teknisiä ominaisuuksia, kuten fysiikkamallinnuksen, kuvanlaadun parantamisen ja äänenjäljityksen. Se on käytössä useissa suosituissa peleissä, kuten Half-Life -, Counter-Strike- ja Portal pelisarjoissa. Source engine on saanut laajan suosion pelialan yhteisössä ja se on yksi tunnetuimmista pelimoottoreista. (Source 2023).

Yksi suosituimmista Source Enginellä tehdyistä peleistä on Counter-Strike: Global Offensive (kuva 6), jolla on lähes miljoona päivittäistä pelaajaa. (Counter-Strike: Global Offensive 2023).

Kuva 6. Counter-Strike: Global Offensive -pelin näyttökuva (Source Counter-Strike: Global Offensive 2023).

3.2 Pelimoottorin järjestelmiä

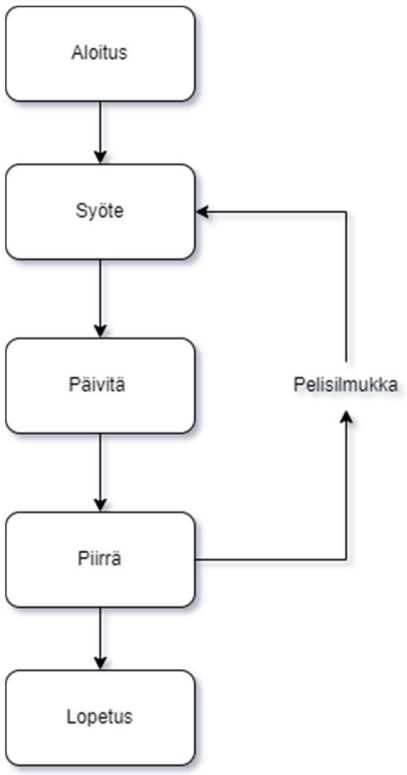
Pelimoottori koostuu useista eri järjestelmistä, jotka toimivat yhdessä luodakseen toimivan kokonaisuuden. Kuvassa 7 on esitelty pelimoottorin ajonaikaisen arkkitehtuurin kokonaisuus.



Kuva 7. Pelimoottorin ajonaikainen arkkitehtuuri (Gregory 2018, 39).

3.2.1 Pelisilmukka

Pelisilmukka tarkoittaa pelimoottorissa tapahtuvaa toistuvaa prosessia, jossa pelin logiikkaa ja grafiikkaa päivitetään jatkuvasti. Kaaviossa 1 on esitettynä tämä yleistasolla. Tämä tapahtuu usein kuvakehys-tasolla, jolloin pelisilmukka ajetaan jokaisen uuden ruudunpäivityksen yhteydessä.



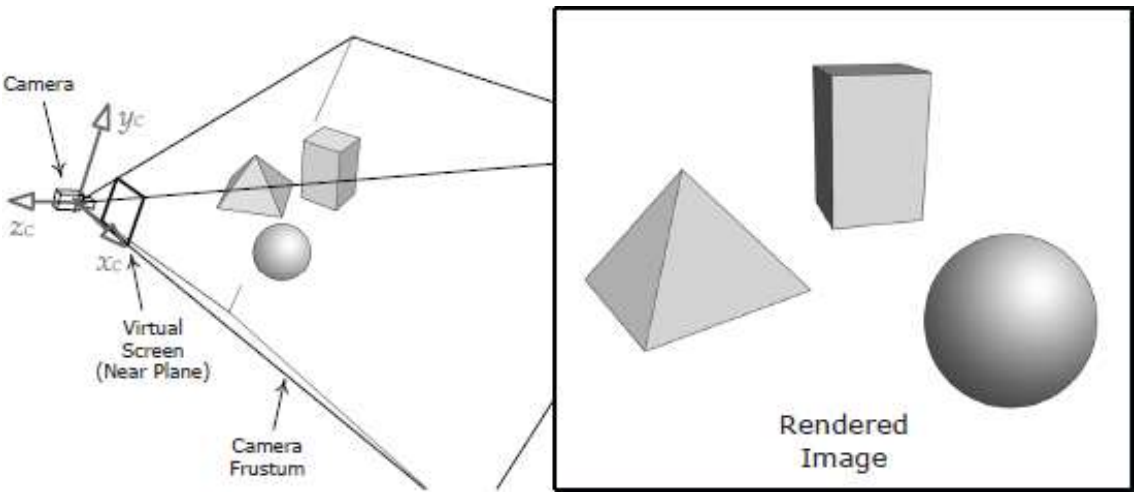
Kaavio 1. Pelisilmukan vuokaavio.

Pelisilmukassa tapahtuu usein monia erilaisia toimintoja, kuten pelaajan ja muiden pelin objektien sijainnin päivitystä, törmäysten tarkistusta, äänien ja musiikin toistamista, sekä grafiikan piirtämistä ruudulle. Pelisilmukka on tärkeä osa pelimoottoria, sillä se ohjaa pelin toimintaa ja mahdollistaa pelin sujuvan toiminnan. (Gregory 2018, 526.)

3.2.2 Kuvanpiirtojärjestelmä

Kuvanpiirtojärjestelmä on osa pelimoottoria, joka vastaa pelin näyttämisestä ruudulla. Se huolehtii siitä, että pelin grafiikka näkyy oikein ja että se päivittyy ruudulla nopeasti. Pelin kuvanpiirtojärjestelmä luo pelin maailman ja sen elementit graafisessa muodossa ja piirtää ne ruudulle. Se käyttää usein erilaisia tekniikoita, kuten 3D-mallinnusta ja tekstuureja, jotta pelin näyttäminen ruudulla olisi realistisempaa. Kuvanpiirtojärjestelmän tehtävä on myös huolehtia siitä, että pelin grafiikka päivittyy ruudulla tarpeeksi nopeasti, jotta peli tuntuu sulavalta ja reagoi nopeasti pelaajan toimiin.

Virtuaalinen pelimaailman näkymä kuvataan, yleensä matemaattisessa muodossa esitettävien 3D-pintojen avulla. Virtuaalikamera sijoitetaan ja suunnataan tuottamaan haluttu näkymä kohtauksesta. (Gregory 2018, 622.) Kuvassa 8 esitetään tämä menetelmä.



Kuva 8. Lähes kaikkien 3D-tietokonegrafiikkateknologioiden käyttämä korkean tason renderöintimenetelmä (Gregory 2018, 623).

3.2.3 Animaatiojärjestelmä

Animaatiojärjestelmä on osa pelimoottoria, joka vastaa hahmojen ja ympäristön animoinnista, eli liikkeen ja ilmeiden esittämisestä pelissä. Se käyttää usein määriteltyjä animaatioita, jotka on tallennettu erillisiin tiedostoihin ja luotu 3D-mallinnuksen avulla. Animaatiojärjestelmän avulla voidaan luoda esimerkiksi hahmojen liikekiertoa, kävely- ja juoksuanimointeja sekä erilaisia ilmeitä ja eleitä. (Kuva 8.).

Kuva 9. Sarja pelihahmon juoksukuvia (Gregory 2018, 722).

Animaatiojärjestelmä voidaan myös yhdistää pelimoottorin fysiikkajärjestelmään, jotta hahmojen liikkuminen ja törmäykset pelissä tuntuisivat realistisilta (Gregory 2018, 721).

3.2.4 Fysiikkajärjestelmä

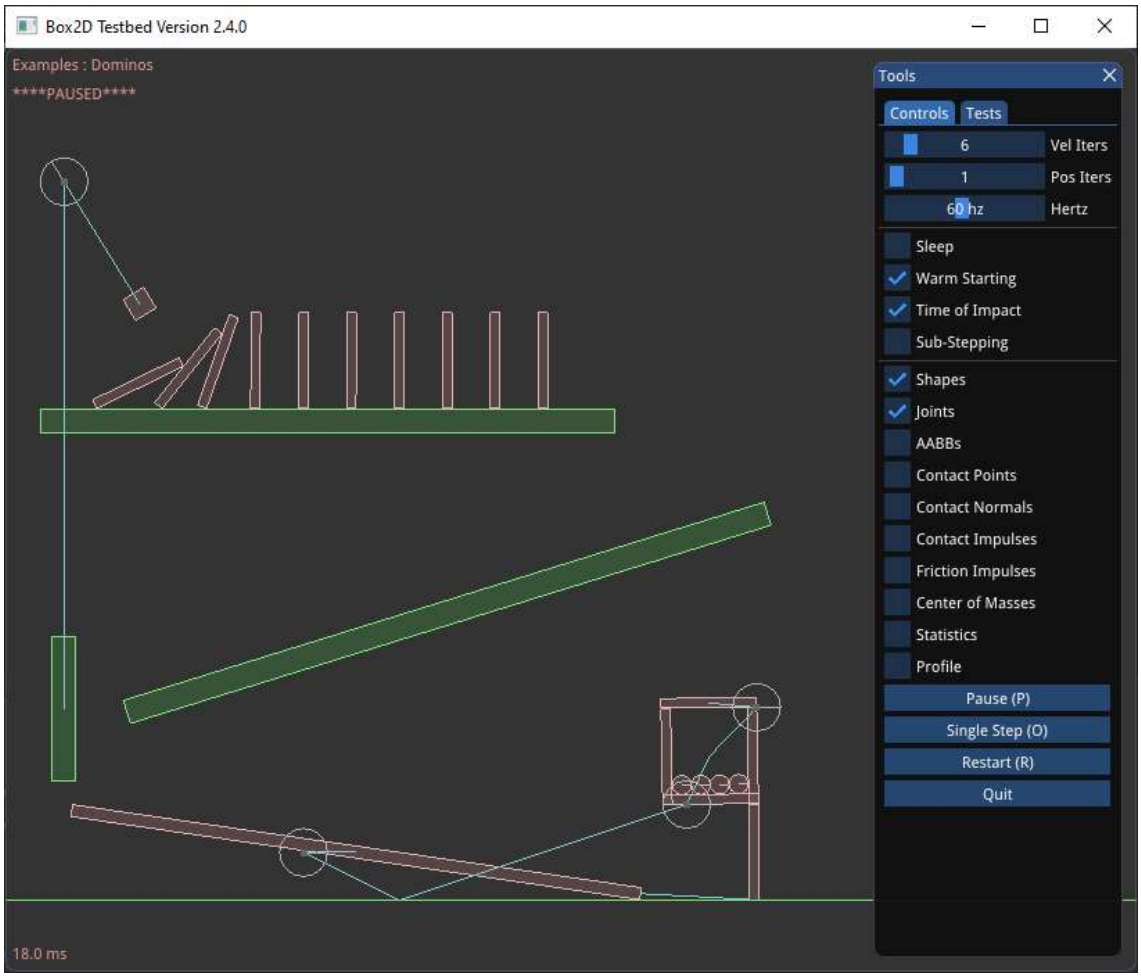
Pelimoottorissa fysiikkajärjestelmä toimii yhdessä törmäysten tunnistuksen kanssa, jotta pelin objektit eivät mene toistensa läpi vaan ne törmäävät toisiinsa ja kimpoavat pois toisistaan.

Pelimoottorin törmäysjärjestelmä on usein läheisesti integroitu fysiikkamoottorin kanssa. Toki fysiikan ala on laaja, ja sitä, mitä useimmat nykypäivän pelimoottorit kutsuvat "fysiikaksi", voidaan kuvailla tarkemmin jäykkäkappaleiden dynamiikan simulaationa. Jäykkä kappale on ihanteellinen, äärettömän kova, ei-muovautuva kiinteä esine. Dynamiikka-termi viittaa siihen, kuinka nämä jäykät kappaleet liikkuvat ja vuorovaikuttavat ajan myötä voimien vaikutuksesta. Jäykkäkappaleiden dynamiikan simulaatio mahdollistaa liikkeen antamisen pelin objekteille erittäin interaktiivisella ja luonnollisen kaaosmaisella tavalla - vaikutus, joka on paljon vaikeampi saavuttaa käyttämällä valmiita animaatioleikkeitä liikuttamaan asioita. (Gregory 2018, 817.)

Dynaamisen simulaation käyttö hyödyntää raskaasti törmäyksen havaitsemisjärjestelmää, jotta voidaan asianmukaisesti simuloida erilaisten objektien fyysisiä käyttäytymismalleja simulaatiossa, kuten kimpoamista toisistaan, liukumista kitkan alla, pyörimistä ja pysähtymistä. Toki törmäyksen havaitsemisjärjestelmää voidaan käyttää erillisenä, ilman dynaamista simulaatiota - monet pelit eivät käytä

lainkaan "fysiikka"-järjestelmää. Mutta kaikilla peleillä, joihin kuuluu objektien liikkuminen kaksi- tai kolmiulotteisessa tilassa, on jonkinlainen törmäyksen havaitsemisjärjestelmä. (Gregory 2018, 817.)

Kuvassa 9 on esitelty Box2D-fysiikkajärjestelmän kokeilualusta, jossa pääsee testaamaan erilaisia ominaisuuksia.



Kuva 10. Box2D-fysiikkajärjestelmän kokeilualusta (Box2D testbed 2023).

3.2.5 Äänijärjestelmä

Äänijärjestelmä pelimoottorissa vastaa äänen toistamisesta ja hallinnoimisesta pelissä. Se voi sisältää erilaisia toimintoja, kuten ääniefektien-, musiikin- ja puheen toistamisen. Äänijärjestelmä voi olla integroitu osa pelimoottoria tai se voi olla erillinen ohjelmisto, joka on integroitu pelimoottoriin.

Nykyajan pelit kietovat pelaajan realistiseen (tai puolirealistiseen mutta tyylikkääseen) virtuaaliympäristöön. Grafiikkamoottorin tehtävänä on toistaa mahdollisimman tarkasti ja uskottavasti se, mitä pelaaja todellisuudessa näkisi, jos hän olisi läsnä tässä virtuaaliympäristössä (pysyen silti pelin taiteellisen tyylin mukaisena). Täsmälleen samalla tavalla äänimoottorin tehtävänä on toistaa tarkasti ja uskottavasti se, mitä pelaaja todellisuudessa kuulisi, jos hän olisi läsnä pelimaailmassa (pysyen silti pelin fiktion ja äänellisen tyylin mukaisena). Ääniohjelmoijat käyttävät nykyään termiä äänentoiston moottori korostaakseen sen monia yhtäläisyyksiä kuvanpiirtojärjestelmän kanssa. (Gregory 2018, 911.)

3.2.6 ECS-järjestelmä

ECS eli Entity Component System on ohjelmiston arkkitehtuurinen malli, jota käytetään usein videopelien kehittämisessä. Se helpottaa koodin uudelleen käytettävyyttä erottamalla datan ja käyttäytymisen. (Terra 2018.)

ECS-järjestelmä koostuu seuraavanlaisista elementeistä:

- Entiteetti eli yksilöllinen tunniste.
- Komponentti, joka koostuu yksinkertaisista datatyypeistä.
- Järjestelmä, joka koostuu funktioista, jotka liittyvät entiteetteihin, joilla on tietyt komponentit.
- Entiteetit voivat sisältää nolla tai enemmän komponentteja.
- Entiteetit voivat muuttaa komponentteja dynaamisesti.

(Terra 2018.)

4 TOTEUTUS

Tässä toteutuksessa on seurattu Robert Wellsin tuottamaa 2D-pelimoottorin kehityssarjaa CPP Game Dev. Muita merkittäviä lähteitä tälle työlle on Raimondas Pupiuksen kirjoittama SFML Game Development By Example kirja sekä The Cherno -nimisen YouTube-kanavan C++- ja Game engine -sarjat, jotka ovat toimineet apuna ja inspiraationa.

Tässä työssä syntyi yli 3000 riviä koodia, joten kaikkia yksityiskohtia ei käsitellä. Sen sijaan keskitytään esittelemään pääjärjestelmien luokkia ja joitain metodeja.

4.1 Kehitysympäristö ja kirjastot

Microsoft Visual Studio 2022 on kehitysympäristö, joka on suunniteltu ohjelmistokehittäjille, jotka haluavat luoda erilaisia ohjelmistoja. Nämä voivat olla sovelluksia, palvelimia, verkkosovelluksia jne. Se tarjoaa laajan valikoiman työkaluja ja ominaisuuksia, jotka auttavat kehittäjiä kirjoittamaan tehokasta, laadukasta koodia nopeasti ja helposti.

C++ ohjelmointikieli on suosittu sen tehokkuuden, monipuolisuuden ja laajennettavuuden vuoksi. Se on yleinen kieli, jota käytetään laajasti ohjelmistokehityksessä, erityisesti sellaisten sovellusten kanssa, joissa suorituskyky ja resurssitehokkuus ovat tärkeitä. Hyvä esimerkki tästä on muun muassa pelikehitys. (C++ Tutorial 2023.)

SFML on lyhenne sanoista Simple and Fast Multimedia Library, joka on kirjasto, joka tarjoaa ohjelmoijille mahdollisuuden luoda monenlaisia multimediaohjelmia, kuten pelejä, animaatioita ja muita graafisia sovelluksia. SFML tarjoaa kehittäjille monia valmiita työkaluja, kuten grafiikka-, ääni-, verkko-ohjelmointi-, järjestelmä- ja ikkunamoduulin. (Gomila nd.)

RapidXML on kirjasto, jota käytetään XML-dokumenttien käsittelyyn. Se on nopea ja kevyt kirjasto, joka on suunniteltu erityisesti suorituskykyyn. RapidXML

tarjoaa kehittäjille monia toimintoja, kuten XML-tiedoston lataamisen, analysoinnin ja muokkaamisen. (RapidXML 2009.)

4.2 Pääohjelma

Pääohjelman tarkoituksena on luoda pelimoottoriobjekti, joka hoitaa tarvittavien järjestelmien päivittämisen ja ohjaamisen (koodi 1).

```
int main()
{
		std::unique_ptr<Engine> coffeeBean = std::make_unique<Engine>();
		coffeeBean->Run();

		return EXIT_SUCCESS;
}
```

Koodi 1. Pääohjelman koodi.

Engine Class objektin *Run()*-metodi (koodi 2) pitää sisällään pelisilmukan, joka hoitaa syötteen tunnistuksen, järjestelmien päivittämisen, piirtämisen ja kuvakehyksen ajan mittauksen.

```
void Engine::Run()
{
		while (IsRunning())
		{
				CaptureInput();
				Update();
				LateUpdate();
				Draw();
				CalculateDeltaTime();
		}
}
```

Koodi 2. Moottorin *Run()*-metodin koodi.

IsRunning()-metodi kutsuu ikkuna -objektin *IsOpen()*-metodia, joka palauttaa tosi arvon, jos ikkunaa ei ole suljettu.

CaptureInput()-metodi kutsuu ”kohtauksen ohjaajan” -metodia *ProsessInput()*, mikä tallentaa pelaajan syötteet.

Update()-metodi kutsuu ”kohtauksen ohjaajan” -metodia *Update()*, mikä päivittää ajossa olevan kohtauksen peliobjektit.

LateUpdate()-metodi kutsuu ”kohtauksen ohjaajan” -metodia *LateUpdate()*, mikä päivittää peliobjektit myöhemmin kuin *Update()*. Tätä tarvitaan joissain tapauksissa esimerkiksi peliobjektin edellisen sijainnin tallentaminen seuraavaa päivityskierrosta varten.

Draw()-metodi kutsuu ikkunaobjektin *BeginDraw()*-metodia piirron alussa, mikä tyhjentää näytön. Tämän jälkeen kohtauksen ohjaaja kutsuu kyseisen kohtauksen piirtometodia, joka piirtää peliobjektit ikkunaobjektin väliaikaismuistiin. Tämän jälkeen ikkunaobjekti kutsuu *EndDraw()*-metodia, mikä piirtää peliobjektit näytölle.

CalculateDeltaTime()-metodi laskee kuluneen ajan edellisestä ruudunpäivityksestä.

Liitteessä 1 on kuvattu ohjelman vuokaavio, jossa on esitetty muun muassa edellä mainittujen toimintojen riippuvuuksia.

4.3 Syötteen tunnistus

Input Class eli syöteluokka (koodi 3) pitää sisällään bittimaskiobjektin, joka on 32-bittinen kokonaislukumuuttuja ja tämä bittimaskiobjekti sisältää metodeja, joiden avulla bittioperaatiota on helpotettu. Tätä bittimaskin arvoa päivitetään *Update()*-metodissa, riippuen siitä onko, jotain näppäintä painettu (koodi 4).

```
enum class Key {
                    None = 0, Left = 1,
                    Right = 2, Up = 3,
                    Down = 4, Esc = 5,
                    LBracket = 6, RBracket = 7,
                    O = 8, E = 9
};
class Input {
public:
                    Input();
                    ~Input();
                    void Update();
                    bool IsKeyPressed(Key keycode);
                    bool IsKeyDown(Key keycode);
                    bool IsKeyUp(Key keycode);
private:
                    Bitmask m_thisFrameKeys;
                    Bitmask m_lastFrameKeys;
};
```

Koodi 3. Input Class eli syöteluokka.

Update()-metodissa (koodi 4.) asetetaan edellisen ruudunpäivityksen syöte olemaan nykyisen ruudunpäivityksen syöte. Tämän jälkeen nykyisen ruudunpäivityksen syötteeseen lisätään painetut näppäimet käyttämällä bittimaski objektin tuomia metodeja.

```
void Input::Update() {
                    m_lastFrameKeys.SetMask(m_thisFrameKeys);
                    m_thisFrameKeys.SetBit((int)Key::Left,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::Left) ||
                                         sf::Keyboard::isKeyPressed(sf::Keyboard::A));

                    m_thisFrameKeys.SetBit((int)Key::Right,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::Right) ||
                                         sf::Keyboard::isKeyPressed(sf::Keyboard::D));

                    m_thisFrameKeys.SetBit((int)Key::Up,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::Up) ||
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::W));

                    m_thisFrameKeys.SetBit((int)Key::Down,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::Down) ||
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::S));

                    m_thisFrameKeys.SetBit((int)Key::Esc,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::Escape));

                    m_thisFrameKeys.SetBit((int)Key::LBracket,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::LBracket));

                    m_thisFrameKeys.SetBit((int)Key::RBracket,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::RBracket));

                    m_thisFrameKeys.SetBit((int)Key::O,
                                        sf::Keyboard::isKeyPressed(sf::Keyboard::O));
}
```

Koodi 4. Input Class -luokan *Update()*-metodi.

4.4 Kohtauksien ohjaaja

Scene Manager Class eli kohtauksien ohjaajan tehtävä on hallinnoida pelin kohtauksia esimerkiksi alkunäyttö, pelivalikko, pelikohtaus. Sekä kutsua näiden kohtauksien syötteen tunnistus-, päivitä-, myöhäinen päivitys- ja piirrä-metodeja (koodi 5).

```
using SceneContainer = std::unordered_map<SceneID, std::shared_ptr<Scene>>;
class SceneManager
{
public:
                        SceneManager();
                        void ProcessInput();
                        void Update(float dt);
                        void LateUpdate(float dt);
                        void Draw(Window& window);
                        SceneID AddScene(std::shared_ptr<Scene> scene);
                        void SwitchTo(SceneID id);
                        void Remove(SceneID id);
private:
                        SceneContainer m_scenes;
                        std::shared_ptr<Scene> m_currentScene;
                        SceneID m_insertedSceneID;
};
```

Koodi 5. Scene Manager Class eli kohtauksien ohjaajaluokka.

4.4.1 Kohtaus

Scene Class eli kohtaus on luokka, mikä pitää sisällään vain virtuaalisia metodeja, joita jokainen kohtaus luokka perii ja yli kirjoittaa (koodi 6). Esimerkki tämän

```
using SceneID = unsigned int;
struct SharedContext;
class Scene
{
public:
                        virtual void OnCreate() = 0;
                        virtual void OnDestroy() = 0;
                        virtual void OnActivate() {};
                        virtual void OnDeactivate() {};
                        virtual void ProcessInput() {};
                        virtual void Update(float dt) {};
                        virtual void LateUpdate(float dt) {};
                        virtual void Draw(Window& window) {};
};
```

Koodi 6. Scene Class eli kohtaus luokka, josta kohtaukset perivät yhteisen rajapinnan.

4.4.2 Peli kohtaus

Game Scene Class eli pelikohtaus on luokka, joka luo pelimaailman, pelaajan ja ylläpitää tämän maailman päivittämisen ja piirtämisen (koodi 7).

```
class Scene_Game : public Scene
{
public:
                        Scene_Game(ResourceManager<sf::Texture>& textureManager, Window& window);
                        void OnCreate() override;
                        void OnDestroy() override;
                        void ProcessInput() override;
                        void Update(float dt) override;
                        void Draw(Window& window) override;
                        void LateUpdate(float dt) override;
private:
                        void CreateMap(const sf::Vector2f& offset);
                        void CreatePlayer(const sf::Vector2f& pos);
                        Input m_input;
                        ResourceManager<sf::Texture>& m_textureManager;
                        EntityManager m_entities;
                        TileMapParser m_mapParser;
                        Window& m_window;
                        SharedContext m_context;
                        sf::FloatRect m_mapBounds;
};
```

Koodi 7. Game Scene eli pelikohtaus luokka.

OnCreate() -metodissa luodaan pelimaailma ja pelaaja. Lisäksi pelikohtaukseen lisätään kamera peliobjekti (koodi 8).

```
void Scene_Game::OnCreate()
{
                        CreateMap(sf::Vector2f(32 * 3, 32));
                        CreatePlayer(sf::Vector2f(400, 700));

                        auto mainCamera = std::make_shared<Entity>(&m_context);
                        mainCamera->AddComponent<C_Camera>();
                        mainCamera->transform->SetPosition(sf::Vector2f(m_mapBounds.width, m_mapBounds.height));

                        sf::View gameView = sf::View(m_mapBounds);
                        gameView.setSize(sf::Vector2f(m_mapBounds.width * 2, m_mapBounds.height * 2));

                        m_window.SetView(gameView);
                        m_entities.Add(mainCamera);

}
```

Koodi 8. Game Scene -luokan *OnCreate()*-metodi.

4.5 Laattakartan jäsentäjä

Tile Map Parser Class (koodi 9.) eli laattakartan jäsentäjä luokka lukee tmx -tiedostosta Tiled ohjelman avulla luodun maailman tiedot ja luo tämän datan avulla maailman pelielementit.

```
struct TileSheetData {
                        int textureId{};
                        sf::Vector2u imageSize;
                        int columns{};
                        int rows{};
                        sf::Vector2u tileSize;
};

struct Layer {
                        std::vector<std::shared_ptr<Tile>> tiles;
                        bool isVisible;
};

using MapTiles = std::vector<std::pair<std::string, std::shared_ptr<Layer>>>;
using TileSet = std::unordered_map<unsigned int, std::shared_ptr<TileInfo>>;
using TileSheets = std::map<int, std::shared_ptr<TileSheetData>>;

class TileMapParser {
public:
                        TileMapParser(ResourceManager<sf::Texture>& textureManager, SharedContext& context);
                        std::vector<std::shared_ptr<Entity>> Parse(const std::string& file, const sf::Vector2i& offset, sf::FloatRect& mapSize);
private:
                        std::shared_ptr<TileSheets> BuildTileSheetData(rapidxml::xml_node<>* rootNode);
                        std::shared_ptr<MapTiles> BuildMapTiles(rapidxml::xml_node<>* rootNode);
                        std::pair<std::string, std::shared_ptr<Layer>> BuildLayer(rapidxml::xml_node<>* layerNode, std::shared_ptr<TileSheets> ti-
leSheets);

                        ResourceManager<sf::Texture>& m_textureManager;
                        SharedContext& m_context;
};
```

Koodi 9. Tile Map Parser Class eli laattakartan jäsentäjäluokka.

4.6 Pelielementtien ohjaaja

Entity Manager Class eli pelielementtien ohjaajaluokan (koodi 10.) tehtävä on hallinnoida kaikki kohtaukseen lisätyt peliobjektit.

```
using EntityContainer = std::vector<std::shared_ptr<Entity>>;

class EntityManager
{
public:
                        void Add(std::shared_ptr<Entity> entity);
                        void Add(std::vector<std::shared_ptr<Entity>>& entities);
                        void Update(float dt);
                        void LateUpdate(float dt);
                        void Draw(Window& window);
                        void ProcessNewEntities();
                        void ProcessRemovals();
private:
                        EntityContainer m_entities;
                        EntityContainer m_newEntitites;
                        S_Drawable m_drawables;
                        S_Collidable m_collidables;
};
```

Koodi 10. Entity Manager Class eli pelielementtien ohjaajaluokka.

4.6.1 Pelielementti

Entity Class eli pelielementtiluokka pitää sisällään metodeja, joilla pelielementtiin voidaan lisätä komponentteja, poistaa komponentteja sekä päivittää pelielementtiä (koodi 11).

```
class Entity
{
public:
                    Entity(SharedContext* context);
                    void Awake();
                    void Start();
                    void Update(float deltaTime);
                    void LateUpdate(float deltaTime);
                    void Draw(Window& window);
                    void OnCollisionEnter(std::shared_ptr<C_BoxCollider> other);
                    void OnCollisionStay(std::shared_ptr<C_BoxCollider> other);
                    void OnCollisionExit(std::shared_ptr<C_BoxCollider> other);
                    std::shared_ptr<C_Drawable> GetDrawable();
                    bool IsQueuedForRemoval();
                    void QueueForRemoval();
                    template <typename T>
                    std::shared_ptr<T> AddComponent(){...};
                    template <typename T> std::shared_ptr<T>
                    GetComponent(){...};
                    std::shared_ptr<C_Transform> transform;
                    std::shared_ptr<C_InstanceID> instanceID;
                    SharedContext* context;
private:
                    std::vector<std::shared_ptr<Component>> m_components;
                    std::shared_ptr<C_Drawable> m_drawable;
                    std::vector<std::shared_ptr<C_Collidable>> m_collidables;
                    bool m_queuedForRemoval;
};
```

Koodi 11. Entity Class eli pelielementtiluokka.

4.6.2 Komponentti

Component Class eli komponentti on luokka, joka antaa jokaiselle komponentille yhteisen rajapinnan (koodi 12).

```
class Entity;
class Component
{
public:
                        Component(Entity* owner) : owner(owner)                 {}
                        virtual void Awake() {};
                        virtual void Start() {};
                        virtual void Update(float dt) {};
                        virtual void LateUpdate(float dt) {};
                        Entity* owner;
};
```

Koodi 12. Component Class eli komponenttiluokka, josta komponentit perivät yhteisen rajapinnan.

Esimerkkinä on sijainti komponentti, jonka tehtävänä on ylläpitää pelielementin sijaintitieto (koodi 13).

```
class C_Transform : public Component
{
public:
                        C_Transform(Entity* owner);
                        void LateUpdate(float dt) override;
                        void SetPosition(float x, float y);
                        void SetPosition(sf::Vector2f position);
                        void AddPosition(float x, float y);
                        void AddPosition(sf::Vector2f position);
                        void SetStatic(bool isStatic);
                        bool IsStatic() const;
                        void SetX(float x);
                        void SetY(float y);
                        void AddX(float x);
                        void AddY(float y);
                        const sf::Vector2f& GetPosition() const;
                        const sf::Vector2f& GetPreviousFramePosition() const;
private:
                        sf::Vector2f m_position;
                        sf::Vector2f m_previousFramePosition;
                        bool m_isStaticTransform;
};
```

Koodi 13. Transform Class eli sijaintiluokka antaa pelielementille sijainti komponentin.

4.7 Resurssien ohjaaja

Resource Manager Class eli resurssien ohjaajan (koodi 14.) tehtävä on luoda yhteinen tietosäilö eri resursseille mm. tekstuureille. Näin monet eri pelielementit pystyvät käyttämään tätä yhtä resurssia eikä jokaisen pelielementin tarvitse ladata tiedostosta kyseistä resurssia.

```
template<typename T>
class ResourceManager
{
public:
                    int Add(const std::string& filepath){...};
                    void Remove(int id){...};
                    std::shared_ptr<T> Get(int id){...};
                    bool Has(int id){...};
private:
                    int m_currentID{};
                    std::map<std::string, std::pair<int, std::shared_ptr<T>>> m_resources;
};
```

Koodi 14. Resource Manager Class eli resurssien ohjaaja luokka.

4.8 Piirtojärjestelmä

Drawable System Class eli piirtojärjestelmä luokka (koodi 15.) tehtävänä on piirtää ne pelielementit, joihin on lisätty piirrettävä komponentti.

```
class S_Drawable
{
public:
                    void Add(std::vector<std::shared_ptr<Entity>>& entities);
                    void ProcessRemovals();
                    void Draw(Window& window);
private:
                    static bool LayerSort(std::shared_ptr<C_Drawable> a, std::shared_ptr<C_Drawable> b);
                    void Add(std::shared_ptr<Entity> entity);
                    void Sort();
                    std::map<DrawLayer, std::vector<std::shared_ptr<C_Drawable>>> m_drawables;
};
```

Koodi 15. Drawable System Class eli piirtojärjestelmä luokka.

4.9 Törmäysjärjestelmä

Collision System Class eli törmäysjärjestelmä luokka (koodi 16.) tehtävänä on hallinnoida pelielementtien törmäyksiä ja törmäysten tunnistusta, joille on asetettuna laatikkotörmäytin.

```
class S_Collidable
{
public:
                    S_Collidable();
                    void Add(std::vector<std::shared_ptr<Entity>>& entities);
                    void ProcessRemovals();
                    void UpdatePositions(std::vector<std::shared_ptr<Entity>>& entities);
                    void Resolve();
                    void Update();
private:
                    void ProcessCollidingObjects();
                    std::unordered_map<CollisionLayer, Bitmask, EnumClassHash> m_collisionLayers;
                    std::unordered_map<CollisionLayer, std::vector<std::shared_ptr<C_BoxCollider>>, EnumClassHash> m_collidables;
                    std::unordered_set<std::pair<std::shared_ptr<C_BoxCollider>, std::shared_ptr<C_BoxCollider>>,
                                        ComponentPairHash> m_objectsColliding;
                    Quadtree m_collisionTree;
};
```

Koodi 16. Collision System Class eli törmäysjärjestelmä.

4.9.1 AABB-laatikkotörmäytin

Tässä törmäysjärjestelmässä käytettään yksinkertaista AABB eli Axis-Aligned Bounding-Box -tyyppistä törmäyksen tunnistusta. Koodissa 17. on esitettynä kahden laatikon törmäyksen tunnistamisen esimerkki.

```
bool checkCollision(const sf::FloatRect& a, const sf::FloatRect& b)
{
                    // If any of the sides from A are outside of B
                    if (a.left + a.width <= b.left)
                                        return false;

                    if (b.left + b.width <= a.left)
                                        return false;

                    if (a.top + a.height <= b.top)
                                        return false;

                    if (b.top + b.height <= a.top)
                                        return false;

                    // If none of the sides from A are outside B
                    return true;
}
```

Koodi 17. Yksinkertainen esimerkki AABB-tyyppisestä törmäyksentunnistuksesta.

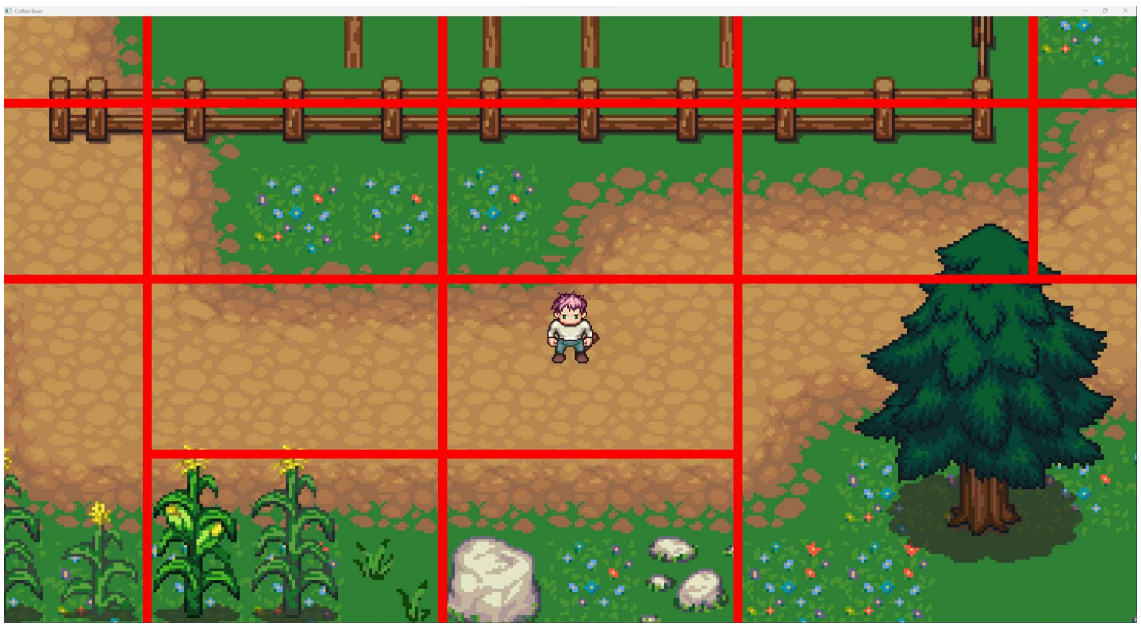
4.9.2 Nelipuu

Quad Tree eli nelipuu on datastruktuuri, jossa kaksiulotteinen tila jaetaan neljään laatikkoon. (Quad Tree, 2022)

1. Jaa kaksiulotteinen tila neljään laatikkoon.
2. Jos laatikko sisältää kohdepisteen, luodaan lapsinelipuu tästä laatikosta.
3. Jos laatikko ei sisällä kohdepisteitä, ei luoda lapsinelipuuta.
4. Käydään läpi rekursiivisesti kaikki lapsinelipuut.

Näin kaksiulotteinen tila saadaan jaettua eri laatikoihin, joista pystytään nopeasti tarkistamaan, pystyykö kaksi kohdepistettä olemaan tekemisissä toisiinsa.

Tässä työssä nelipuuta käytetään muun muassa törmäysjärjestelmän jakamiseen pienempiin alueisiin, jolloin tarkistettavien pelielementtien määrä vähenee huomattavasti. Kuvassa 10 on piirrettynä törmäysnelipuun ääriviivat.



Kuva 11. Quad Tree eli nelipuu törmäysjärjestelmässä.

5 TESTAUS

Tässä luvussa kerrotaan, kuinka animaatio-, törmäys- ja maailman latausjärjestelmää testattiin. Tämän luvun animaation- ja törmäyksien testaus kohdissa esitetyt taideresurssit ovat Liberated Pixel Cup (Opengamesart nd) kilpailusta.

5.1 Animaatiojärjestelmän testaus

Animaatiojärjestelmän testaamiseksi, luotiin animaationtestaus kohtaus, jossa jokainen kuvakehys tallennettiin kuvaksi. Kuvassa 11 on jousipyssyllä ampumisen animaatio kuvakehykset vierekkäin. Tämän testauksen pohjalta voidaan todeta, että animaatiojärjestelmä toimii.

Kuva 12. Jousipyssyllä ampumisen animaatio kuvakehykset.

5.2 Törmäyksien testaus

Kuvassa 12. on esitettynä läpinäkyvän punaisella maailman laatikot, joihin on lisätty laatikkotörmäytin. Lisäksi kuvassa 12 näkyy pelaajahahmon ja maailman törmäytin laatikoiden välinen interaktio piirrettynä punaisilla ääriviivoilla.

Kuva 13. Törmäyslaatikot.

Kuvassa 13 on esitettynä pelaajan ampuneen nuolen törmäyslaatikon ääriviivat sen vuorovaikuttaessa maailman törmäyslaatikoihin.

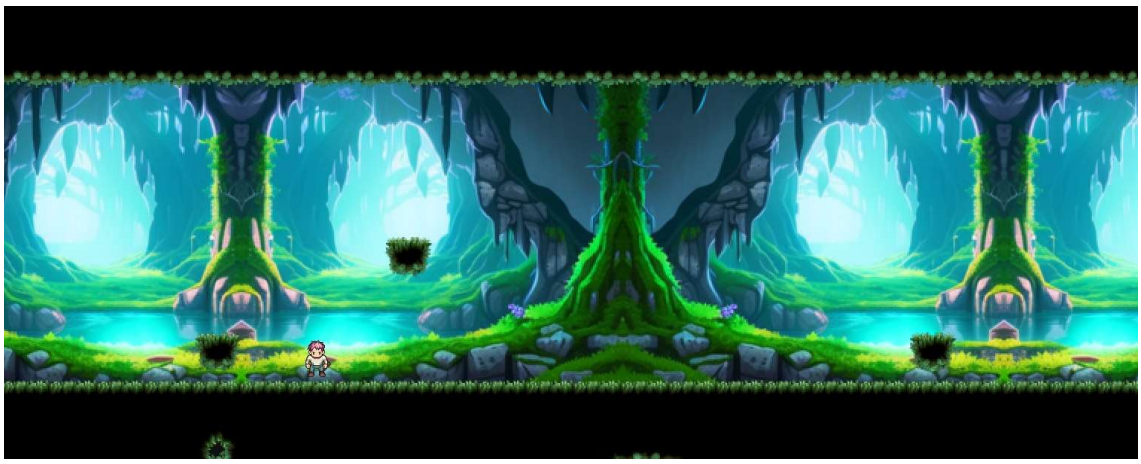
Kuva 14. Nuolen törmäyslaatikot

Näiden törmäysjärjestelmän testauksien perusteella pystyttiin toteamaan, että törmäykset toimivat odotetusti. Pelaaja ei pysty kävelemään sellaisen alueen läpi, johon oli asetettu laatikkotörmäytin sekä nuolet pysähtyvät kohdatessaan laatikkotörmäyttimen.

5.3 Itsetekoisen maailman latauksen testaus

Maailman latauksen testaamista varten luotiin oma maailma käyttämällä Tiled ohjelmaa. Taideresursseina käytettiin Maootin luomaa 2D-resurssipakkausta (Mossy Cavern 2023) ja taustataide luotiin käyttämällä Hotpot-tekoälytaidegeneraattoria (AI Art Generator 2023).

Kuvassa 15 on kuvakehys itsetekoisesta maailmasta, joka ladattiin peliin käyttämällä pelimoottorin laattakartan jäsentäjää. Tästä voidaan todeta, että maailman lataus toimii odotetusti.



Kuva 15. Itsetekoisen maailma ajossa pelimoottorissa

6 POHDINTA

Tässä työssä saatiin aikaan toimiva pelimoottorin malli, joka mahdollistaa Tiled-ohjelmalla luotujen pelimaailmojen lataamisen, hahmojen animaatioiden rakentamisen spritesheeteistä, erilaisten kohtausten muodostamisen pelitasoille ja toiminnallisuuden laajentamisen komponenttien avulla. Työhön ei kuitenkaan sisällytetty varsinaista peliä työmäärän vuoksi.

Pelimoottori on laaja ja monipuolinen käsite, ja tämän työn avulla päästiin vain pintaa raapaisemaan. Siksi työtä voisi jatkokehittää useaan suuntaan. Esimerkiksi käyttöliittymän lisääminen tähän pelimoottoriin tekisi siitä käyttäjäystävällisemmän. Käyttöliittymän toteuttamiseen voisi hyödyntää esimerkiksi Dear ImGui -kirjastoa (Dear ImGui 2023).

Työn ECS-järjestelmän voisi muuttaa käyttämään EnTT-kirjastoa (EnTT 2023), mikä toisi mukanaan tehokkaamman ja todellisen ECS-järjestelmän. Nykyinen ECS-järjestelmä ei noudata ECS-arkkitehtuuria, mutta tarjoaa helposti lähestyttävän käyttöliittymän komponenttien lisäämiseen ja niiden käsittelyyn.

Lisäksi työhön voisi sisällyttää Box2D-fysiikkamoottorin (Box2D 2023), jolloin fysikaaliset simulaatiot ja törmäysten hallinta olisivat tehokkaampia ja monipuolisempia. Tällä hetkellä pelimoottorissa ei ole fysiikkamoottoria, vaan ainoastaan AABB-törmäyksen tunnistus on toteutettu.

LÄHTEET

Mossy Cavern. 2023. itch.io. Verkkosivu. Viitattu 19.3.2023
https://maaot.itch.io/mossy-cavern

The Brown Box, 1967-68. 2006. Verkkosivu. Viitattu 5.3.2023
https://americanhistory.si.edu/collections/search/object/nmah_1301997

Box2D. 2023. Box2D Verkkosivu. Viitattu 12.3.2023
https://box2d.org/

Box2D testbed. 2023. Box2D. Verkkosivu. Viitattu 12.3.2023
https://box2d.org/documentation/md__d_1__git_hub_box2d_docs_testbed.html

C++ Tutorial. 2023. W3Schools. Verkkosivu. Viitattu 11.3.2023
https://www.w3schools.com/cpp/

Counter-Strike: Global Offensive. 2023. STEAM CHARTS. Verkkosivu. Viitattu 5.3.2023
https://steamcharts.com/app/730

Cryengine. 2023. Cryengine. Verkkosivu. Viitattu 5.3.2023
https://www.cryengine.com/

Cryengine Crysis 3. 2023. Cryengine. Viitattu 5.3.2023
https://www.cryengine.com/showcase/view/crysis-3

Dear ImGui. 2023. GitHub. Verkkosivu. Viitattu 12.3.2023
https://github.com/ocornut/imgui

Lowood, H. 2020. Doom. Verkkosivu. Viitattu 5.3.2023
https://www.britannica.com/topic/Doom

Terra, J. 2023. Entity Component System: An Introductory Guide. simplilearn. Verkkosivu. Viitattu 15.4.2023.
https://www.simplilearn.com/entity-component-system-introductory-guide-article

EnTT. 2023. GitHub. Verkkosivu. Viitattu 12.3.2023
https://github.com/skypjack/entt

Lowood, H. 2023. Game Engines and Game History. Verkkosivu. Viitattu 5.3.2023
https://www.kinephanos.ca/2014/game-engines-and-game-history/

Gregory, J. 2018. Game engine architecture. 3. Painos. Boca Raton. CRC Press.

AI Art Generator. 2023. Hotpot. Verkkosivu. Viitattu 19.3.2023
https://hotpot.ai/art-generator

Opengamesart. nd. The Liberated Pixel Cup. Verkkosivu. Viitattu 15.4.2023
https://lpc.opengameart.org/

Pupius, R. 2015. SFML Game Development By Example. 1. Painos. Birmingham. Packt Publishing Ltd.

Quad Tree. 2022. Geeks for Geeks. Verkkosivu. Viitattu 11.3.2023
https://www.geeksforgeeks.org/quad-tree/

RapidXML. 2009. Sourceforge. Verkkosivu. Viitattu 11.3.2023
https://rapidxml.sourceforge.net/

Gomila, L. n.d. SFML-DEV. Verkkosivu. Viitattu 5.3.2023
https://www.sfml-dev.org/index.php

Source. 2023. Valve Developer Community. Verkkosivu. Viitattu 5.3.2023
https://developer.valvesoftware.com/wiki/Source

Source Counter-Strike: Global Offensive. 2023. Valve Developer Community. Verkkosivu. Viitattu 5.3.2023
https://developer.valvesoftware.com/wiki/Counter-Strike:_Global_Offensive

Steam Charts. 2023. Steam. Verkkosivu. Viitattu 19.3.2023
https://store.steampowered.com/charts/mostplayed/

The First Video Game?. 2008. Brookhaven National Laboratory. Verkkosivu. Viitattu 12.3.2023
https://www.bnl.gov/about/history/firstvideo.php

Tyler, D. 2023. How to Choose the Best Video Game Engine. gamedesigning.org. Verkkosivu. Viitattu 19.3.2023
https://www.gamedesigning.org/career/video-game-engines/#The-25-Most-Popular-Game-Engines

Unity. 2023. Unity. Verkkosivu. Viitattu 5.3.2023
https://unity.com/

Unity Escape from Tarkov. 2023. Unity. Verkkosivu. Viitattu 5.3.2023
https://unity.com/madewith/escape-from-tarkov

Video Game History. 2022. History.com. Verkkosivu. Viitattu 12.3.2023
https://www.history.com/topics/inventions/history-of-video-games

Wells, R. 2018. Game Dev Series. That Games Guy. Verkkosivu. Viitattu 5.3.2023
https://thatgamesguy.co.uk/cpp-game-dev-0/

LIITTEET

Liite 1. Pelimoottorin pelisilkkaa